Certificación PCAP

Lección 6: Módulos y paquetes, excepciones, generadores y cierres y procesado de ficheros

eip ESCUELA INTERNACIONAL POSGRADOS

ÍNDICE

Lección 6: Módulos y paquetes, excepciones, generadores y cierres y procesado de ficheros

1. Objetivos

En esta última lección, vamos a ver diferentes temas como son los módulos y paquetes, las excepciones, los generadores y cierres, las funciones lambda y cómo procesar ficheros de texto y binarios, centrándonos principalmente en aquellos aspectos que suelen aparecer de forma común en el examen de la certificación PCAP.

2. Módulos y Paquetes

Cualquier código que se desarrolla tiende a modificarse y crecer a lo largo del tiempo. De hecho, el código que no tiene esta tendencia probablemente sea inutilizable o esté abandonado. Un código real y ampliamente utilizado se desarrolla continuamente adaptándose a las nuevas demandas y necesidades de los usuarios.

Conforme el código va creciendo se hace más grande y su mantenimiento se hace más difícil. Por ejemplo, la búsqueda de errores siempre es más fácil cuando el código es más pequeño.

Además, conforme el código va creciendo y se va haciendo más grande, es habitual, que participen más desarrolladores en él, por lo que será necesario dividirlo en muchas partes, para que estos desarrolladores puedan trabajar implementándolo en paralelo.

Por lo tanto, es necesario que Python proporcione una herramienta para dividir una pieza de software en partes separadas pero cooperantes y esta herramienta son los **módulos**.

Un módulo se define como un **fichero que contiene definiciones y declaraciones de Python y que luego se puede importar y usar cuando sea necesario**.

Los módulos se identifican por su nombre por lo que, si se desea utilizar cualquier módulo, se necesita conocer su nombre. Además, cada módulo consta de entidades que pueden ser funciones, variables, constantes, clases y objetos. Si se sabe cómo acceder a un módulo concreto se puede utilizar cualquiera de estas entidades.

Hay dos formas de trabajar con módulos:

- Trabajar utilizando módulos ya existentes, escritos por otra persona. Probablemente esto sea lo más común. El programador será el **usuario** del módulo. Por ejemplo, Python proporciona una cantidad bastante grande de módulos. Todos estos módulos, junto con las funciones integradas, forman la **Biblioteca estándar de Python**.
- Trabajar creando un nuevo módulo, ya sea para su uso propio o para facilitar la vida de otros programadores. El programador será el **proveedor** del módulo.

2.1. Importando un módulo

Para que un módulo sea utilizable y podamos utilizar sus entidades es necesario **importarlo**. La importación de un módulo se realiza mediante una instrucción llamada **import**.

Con import se puede importar el módulo de diferentes maneras y dependiendo de cómo se importe dependerá cómo se usan las entidades del módulo importado.

En los diferentes ejemplos y explicaciones vamos a trabajar con diferentes módulos. Uno de ellos es el módulo **math** (https://docs.python.org/es/3/library/math.html). Este módulo proporciona acceso a las funciones matemáticas definidas en el estándar de C. Otro de estos módulos es el módulo sys (https://docs.python.org/es/3/library/sys.html?highlight=sys#module-sys). Este módulo provee acceso a algunas variables usadas o mantenidas por el intérprete y a funciones que interactúan fuertemente con el intérprete.

La forma más sencilla de importar un módulo es utilizar la instrucción de importación de la siguiente manera:

import modulo

```
1.      >>> import math
2.
```

Como se puede observar esta cláusula contiene la palabra reservada import y en nombre del módulo que se va a importar. La instrucción puede colocarse en cualquier parte del código, pero debe colocarse **antes del primer uso de cualquiera de las entidades del módulo**.

Si se desea importar más de un módulo se puede repetir esta cláusula tantas veces como sea necesario:

```
1.      >>> import math
2.      >>> import sys
3.
```

O se puede poner listando los módulos después de la palabra import:

```
>>> import math, sys
```

Con esta instrucción importaremos los dos módulos, primero importando el módulo math y posteriormente el módulo sys.

Con este tipo de importación si queremos utilizar cualquiera de las entidades de unos de estos módulos importados, podremos acceder a ellas y utilizarlas de la siguiente forma:

```
>>> import math

>>> print(math.pi)
3.141592653589793

>>> print(math.radians(90))
1.57079632679489

>>> print(math.sin(math.radians(90)))
 1.0
```

Como podemos observar para acceder a cualquiera de las entidades del módulo math, debemos poner el nombre del módulo (math), un punto y el nombre de la entidad. En este ejemplo hemos utilizado tres entidades, la primera de ellas “pi” es una constante de contiene el valor del número matemático pi. La segunda de ellas, “radians”, es una función que convierte de grados a radians. Y la tercera de ellas, “sin”, es otra función que calcula el seno de x radianes.

Una vez hemos visto cómo trabajar con las entidades del módulo con este tipo de importación, es necesario introducir un nuevo concepto para entender mejor la importación: el **namespace**.

Un namespace es un espacio (en un contexto no físico) en el que existen algunos nombres y los nombres no entran en conflicto entre sí (es decir, no hay dos entidades diferentes con el mismo nombre). Es decir, **dentro de un namespace, cada nombre debe permanecer único**. Esto significa que algunos nombres pueden desaparecer cuando cualquier otra entidad de un nombre ya conocido entra en el namespace. Parece un concepto un poco complicado, pero lo vamos a ir viendo con ejemplos para entenderlo mejor.

Al importar con este método, Python importará su contenido y entidades, pero no ingresan en el namespace del código. Esto significa que podemos tener en nuestro código nuestras propias entidades llamadas sin o pi y no serán afectadas en alguna manera por el import:

```
>>> import math

>>> pi = 5
>>> print(math.pi)
3.141592653589793
>>> print(pi)
5

>>> def sin(a):
        return a + 5
>>> print(math.sin(math.radians(90)))
1.0
>>> print(sin(math.radians(90)))
6.57079632679489
```

En el segundo método de importación, la sintaxis del import señala con precisión qué entidad (o entidades) del módulo se van a poder utilizar en el código:

from modulo import entidad

```
>>> from math import pi

```

En este tipo de importación, la instrucción consta de los siguientes elementos:

- La palabra reservada from.
- El nombre del módulo cuyas entidades se van a importar.
- La palabra reservada import.
- El nombre o lista de nombres de la entidad o entidades que se van a importar al namespace.

Al utilizar esta instrucción, las entidades listadas son las únicas que son importadas del módulo importado, es decir, si intentamos acceder a alguna entidad del módulo que no ha sido listada, el código nos devolverá un error.

Además, los nombres de las entidades importadas pueden ser accedidas dentro del programa. Con este método, por lo tanto, podremos utilizar las entidades importadas de la siguiente manera:

```
>>> from math import pi, radians, sin

 >>> print(pi)
3.141592653589793

>>> print(radians(90))
1.57079632679489

>>> print(sin(radians(90)))
 1.0
```

Como vemos a la hora de utilizar las entidades del módulo ya no es necesario poner "math.entidad" como hacíamos con el anterior método de importación. Es más, si intentamos llamar a una entidad de esta forma, el código nos devolverá un error:

```
>>> from math import pi, radians, sin

 >>> print(math.pi)
Traceback
NameError: name 'math' is not defined
```

Lo mismo pasará si se intenta utilizar una entidad que no se ha definido en la lista de entidades del import:

```
>>> from math import radians, sin

 >>> print(pi)
Traceback
NameError: name 'pi' is not defined
```

Por otra parte, hay que recalcar que con este tipo de import las entidades importadas se importan en el namespace del código. Esto quiere decir que, si tenemos definidas entidades con el mismo nombre antes del import en nuestro código, dichas entidades serán sustituidas con las entidades importadas:

```
>>> pi = 5
>>> def sin(a):
    return a + 5

>>> print(pi)
5
>>> print(sin(1.57079632679489))
6.57079632679489

>>> from math import pi, sin
>>> print(pi)
3.141592653589793
>>> print(sin(1.57079632679489))
1.0
```

Como vemos en el ejemplo, después del import las entidades pi y sin han sido sustituidas por las importadas.

Del mismo modo, si hacemos el import y después definimos en nuestro código entidades con el mismo nombre, las entidades del import serán sustituidas con las entidades definidas:

```
>>> from math import pi, sin
>>> print(pi)
3.141592653589793
>>> print(sin(1.57079632679489))
1.0

>>> pi = 5
>>> def sin(a):
    return a + 5

>>> print(pi)
5
>>> print(sin(1.57079632679489))
6.57079632679489
```

Hay un tercer método para importar módulos similar al segundo método visto, pero mucho más agresivo:

from modulo import *

```
>>> from math import *

```

Con esta instrucción se importarán todas las entidades del módulo indicado y se añadirán al namespace por lo que hay que usarla con precaución ya que es complicado controlar los posibles conflictos de nombres.

```
>>> from math import *
>>> print(pi)
3.141592653589793
>>> print(sin(1.57079632679489))
1.0
```

Por otra parte, si se importa un módulo y se quiere cambiar el nombre del módulo para, por ejemplo, trabajar con el módulo de forma más cómoda, Python proporciona la cláusula **as** que permite realizar este renombrado o **aliasing**.

import modulo as alias

```
>>> import math as m
```

Donde alias es el nuevo nombre con el que voy a utilizar el módulo. Es importante tener en cuenta que después de este renombrado, el nombre original del módulo se volverá inaccesible y ya no podrá ser utilizado.

```
>>> import math as m

>>> print(m.pi)
3.141592653589793

>>> print(math.pi)
Traceback
NameError: name 'math' is not defined
```

Los alias también pueden ser utilizados con el segundo método de importación para renombrar las entidades:

from modulo import entidad1 as alias1, entidad2 as alias2

En este caso pasará lo mismo, una vez renombremos una entidad esta ya no podrá ser utilizada con su nombre original:

```
>>> from math import pi as PI

>>> print(PI)
3.141592653589793

>>> print(pi)
Traceback
NameError: name 'pi' is not defined

```

En este segundo tipo de importación loas alias pueden ser muy útiles para evitar problemas de conflictos:

```
>>> pi = 5
>>> from math import pi as PI
>>> print(pi)
5
>>> print(PI)
3.141592653589793

```

Para terminar con los módulos, vamos a ver la función dir. Esta función devuelve una lista ordenada alfabéticamente la cual contiene todos los nombres de las entidades disponibles en el módulo:

dir(modulo)

```
>>> import math

>>> print(dir(math))
['__doc__', '__loader__', '__name__', '__package__',
'__spec__', 'acos', 'acosh', 'asin', 'asinh', 'atan',
'atan2', 'atanh', 'ceil', 'comb', 'copysign', 'cos',
'cosh', 'degrees', 'dist', 'e', 'erf', 'erfc', 'exp',
'expm1', 'fabs', 'factorial', 'floor', 'fmod', 'frexp',
'fsum', 'gamma', 'gcd', 'hypot', 'inf', 'isclose',
'isfinite', 'isinf', 'isnan', 'isqrt', 'ldexp', 'lgamma',
'log', 'log10', 'log1p', 'log2', 'modf', 'nan', 'perm',
'pi', 'pow', 'prod', 'radians', 'remainder', 'sin',
'sinh', 'sqrt', 'tan', 'tanh', 'tau', 'trunc']
```

Es importante conocer que para utilizar esta función, el módulo debe haberse importado previamente con el primer método de importación visto.

El resumen las tres formas de importar módulos que hemos visto son:

- import modulo -> No incluye las entidades del módulo en el namespace del código. Puedo utilizar todas las entidades del módulo en mi código.
- from modulo import entidad -> Incluye las entidades listadas en el namespace del código. Sólo puedo utilizar las entidades listadas en mi código.
- from modulo import * -> Incluye todas las entidades del módulo en el namespace del código. Puedo utilizar todas las entidades del módulo en mi código. No se recomienda esta forma de importar ya que hace difícil controlar los posibles conflictos y sobreescritura de entidades.

2.2. Paquetes

Como hemos visto un módulo es un contenedor que nos permite empaquetar funciones, constantes, clases y objetos para evitar tener código muy grande. Pero, ¿qué pasa si al final tenemos muchos módulos? ¿Hay alguna forma de agrupar dichos módulos? Efectivamente, Python nos proporciona una herramienta para agrupar nuestros módulos: los **paquetes**.

Imaginemos que tenemos dos módulos creados por nosotros cuyos nombres son mimodulo1 y mimodulo2 y por lo tanto cada uno de ellos estará en un fichero Python diferente: mimodulo1.py y mimodulo2.py. Cada uno de estos ficheros podría tener el siguiente código:

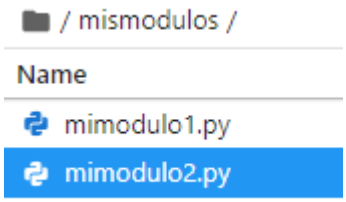
mimodulo1.py

```
1.      print("INICIO MODULO1")
2.      MODULO1_CON = 4
```

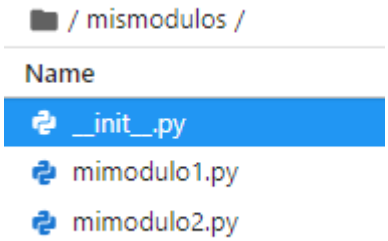
mimodulo2.py

```
1.      print("INICIO MODULO2")
2.      MODULO2_CON = 5
```

Si quisiéramos agruparlos, podríamos guardarlos en el mismo directorio llamado mismodulos:



Pero, ¿cómo podríamos transformar esta estructura de árbol en un paquete de Python? La forma que tiene Python para detectar si un directorio es un paquete es esperando que dicho directorio contenga un fichero llamado **__init__.py**. El contenido de este fichero se puede utilizar para inicializar el paquete ya que es ejecutado cuando alguno de los módulos del paquete se importa. Si no fuese necesario inicializar el paquete, este fichero se puede dejar vacío, pero debe existir.



Por lo que ahora ya podríamos utilizar las entidades de los módulos creados dentro del paquete mismodulos.

```
>>> import mismodulos.mimodulo1
INICIO MODULO 1
>>> print(mismodulos.mimodulo1.MODULO1_CON)
4

```

Como podemos observar para importar el módulo, es necesario indicar el nombre el paquete, punto y el nombre del módulo. Además, al importar el módulo se ejecuta su código, por lo que se ejecuta el print que había declarado mostrado el mensaje “INICIO MODULO 1”. Este código se ejecuta únicamente la primera vez que se importa el módulo. Por lo que, si se importase dos veces, sólo se mostraría una única vez el mensaje. También podemos importar los módulos de un paquete con el segundo y tercer métodos de importación:

```
>>> from mismodulos.mimodulo1 import MODULO1_CON
INICIO MODULO 1
>>> print(MODULO1_CON)
4

```

Por último, indicar que los paquetes también pueden contener subpaquetes y su creación sería creando subdirectorios dentro del paquete. En este caso la creación del fichero __init__.py sería opcional.

3. EXCEPCIONES

Una vez visto cómo crear módulos, importarlos y utilizarlos, así como cómo crear paquetes y utilizarlos, vamos a ver otro tema muy recurrente en las preguntas del examen de certificación PCAP y que son las excepciones.

Como hemos visto a lo largo del curso, es muy normal que al realizar ciertas operaciones el código falle y hemos visto que Python muestra el tipo de error ocurrido. Esto se debe a que cuando al ejecutar el código se intenta hacer algo erróneo, Python detiene el programa y crea un tipo especial de dato, un objeto, llamado **excepción**. Esta acción se llama **lanzar una excepción**.

Una vez se lanza la excepción, Python espera que alguien o algo lo note y haga algo al respecto. Si no se hace nada, el programa termina de forma abrupta y muestra en la pantalla el mensaje de error como hemos visto en ejemplos anteriores. Si, por el contrario, se atiende la excepción y es **manejada** de forma correcta, el programa puede reanudarse y continuar con su ejecución.

Python, por lo tanto, proporciona herramientas para atender de forma correcta las excepciones. Esta herramienta es el bloque try-except. Vamos a ver la forma más sencilla de utilizarla:

```
1.      >>> mi_lista = [1,2,3]
2.      >>> print(mi_lista[3])
3.      Traceback
4.      IndexError: list index out of range
5.
```

Como vemos en el ejemplo, si intentamos acceder a una posición de la lista que no existe, Python nos está devolviendo un error de tipo IndexError y como no hacemos nada para manejarlo, termina la ejecución del programa mostrando dicho error.

```
1.      >>> try:
2.      >>>     mi_lista = [1,2,3]
3.      >>>     print(mi_lista[3])
4.      >>>     print("Terminado")
5.      >>> except:
6.      >>>     print("Capturado el error")
7.      Capturado el error
8.
```

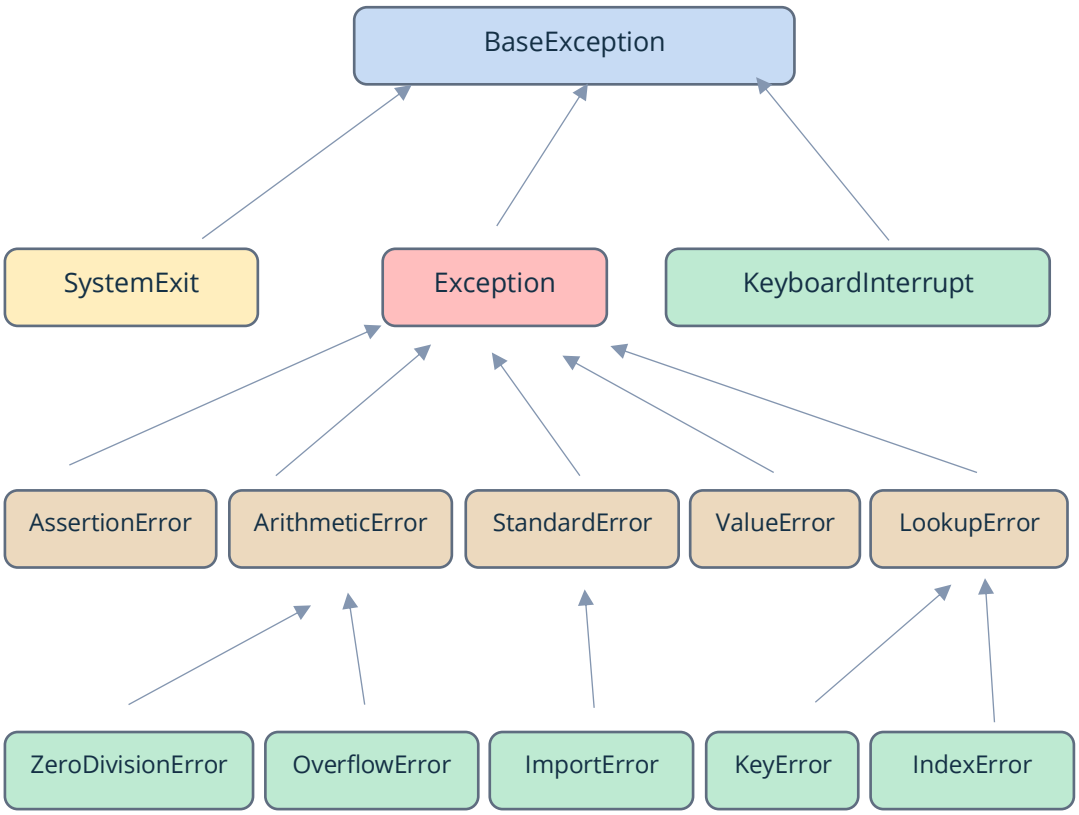
Como vemos en este ejemplo, al principio nos encontramos en la palabra reservada “try” y posteriormente nos encontramos con la palabra reservada “except”. En primer lugar, Python intenta ejecutar el código que hay entre las estas dos palabras clave. Como en la línea 3 el código falla al intentar acceder a una posición de la lista que no existe, Python crea la excepción, para la ejecución y salta a ejecutar las instrucciones definidas en except. Por ello sólo se muestra por pantalla el mensaje definido en la línea 7 y no se muestra el definido en la línea 4.

```
1.      >>> try:
2.      >>>     mi_lista = [1,2,3]
3.      >>>     print(mi_lista[0])
4.      >>>     print(“Terminado”)
5.      >>> except:
6.      >>>     print(“Capturado el error”)
7.      1
8.      Terminado
9.
```

En este otro ejemplo, como no se produce ningún error en el código definido entre las palabras reservadas try-except, se ejecuta todo este código, pero no se ejecuta el código definido dentro de except.

3.1. Excepciones: Jerarquía de clases

Como se ha explicado, cuando se genera un error, Python para la ejecución del programa y crea un objeto excepción. Este objeto excepción puede pertenecer a una clase de tipo excepción. Python 3 define 63 excepciones integradas y que forman una jerarquía de clases.



En este esquema podemos ver algunas de estas clases. Como se puede observar la clase más genérica, la superclase, sería BaseException y de ella heredarían el resto de clases. De cara al examen de certificación es importante conocer esta jerarquía de clases, especialmente conocer cuál son las superclases de aquellas clases de excepción más comunes.

3.2. Excepciones: Capturar una excepción

Una vez vista la jerarquía de clases de las excepciones y la forma más sencilla de capturar una excepción, vamos a continuar profundizando en cómo capturar las excepciones. La forma que hemos visto al inicio de esta sección tiene una gran desventaja, es difícil que conocer qué error ha ocurrido exactamente en el código entre las palabras reservadas try y except. Imaginemos el siguiente ejemplo:

```
>>> try:
>>>     x = int(input("Escriba un número"))
>>>     y = 200 / x
>>> except:
>>>     print("Ha ocurrido un error")
```

En este caso, es imposible conocer si el error se debe a que el usuario ha introducido un valor que no se ha podido convertir a un número entero o si se debe a que la división no se ha podido realizar porque el número introducido ha sido 0.

Python proporciona también herramientas para poder capturar cada una de estas opciones dependiendo el tipo de error lanzado.

```
>>> try:
>>>     x = int(input("Escriba un número"))
>>>     y = 200 / x
>>> except ZeroDivisionError:
>>>     print("No se puede dividir entre 0")
>>> except ValueError:
>>>     print("Debes introducir un número entero")
>>> except:
>>>     print("Alguna otra cosa ha salido mal")
```

En este ejemplo se muestra como con varias sentencias except se puede capturar y tratar de forma personalizada cada uno de los posibles tipos de error que se prevé poder tener.

Si el usuario introduce 0 se mostrará por pantalla "No se puede dividir entre 0". Si el usuario introduce, por ejemplo, "dfd" como no se puede convertir a entero se mostrará por pantalla "Debes introducir un número entero". Y si ocurre cualquier otro tipo de error, como, por ejemplo, si introduce ctrl-c que lanzaría la excepción KeyboardInterrupt, se mostrará "Alguna otra cosa ha salido mal".

Por lo que es importante tener en cuenta que:

- Los bloques “except” son analizados en el mismo orden en que aparecen en el código.
- No se debe usar más de un bloque de excepción con el mismo nombre.
- El número de diferentes bloques except es arbitrario, la única condición es que si utilizas try debes poner al menos un except después de él.
- La palabra reservada except no debe ser empleada sin que le preceda un try.
- Si uno de los bloques except es ejecutado, ningún otro lo será.
- Si ninguno de los bloques except especificados coincide con la excepción lanzada, la excepción permanece sin manejar. Por ejemplo:

```
>>> try:
>>>     y = 200 / 0
>>> except ValueError:
>>>     print(“Valor no correcto”)
Traceback
ZeroDivisionError: división by zero
```

- Si se utiliza un except sin nombre de excepción, tiene que especificarse siempre al final.

Como hemos visto las excepciones cumplen una jerarquía de clases, por lo que, ¿qué pasaría si en el siguiente fragmento de código sustituyésemos la excepción ZeroDivisionError por una clase más general, por ejemplo, ArithmeticError, Exception o BaseException?

```
>>> try:
>>>     y = 200 / 0
>>> except ZeroDivisionError:
>>>     print(“No se puede dividir entre 0”)
No se puede dividir entre 0
```

Lo que pasaría es que como la excepción se capturará con la primera coincidencia y la coincidencia no tiene que especificar exactamente la misma excepción, si no que es suficiente que la excepción sea más general que la lanzada, se capturará igual:

```
>>> try:
>>>     y = 200 / 0
>>> except ArithmeticError:
>>>     print("Error en la operación")
Error en la operación
```

Es más, si en vez de sustituirla la añadimos encima lo que pasará es que al ser la primera ocurrencia, se capturará con ella:

```
>>> try:
>>>     y = 200 / 0
>>> except ArithmeticError:
>>>     print("Error en la operación")
>>> except ZeroDivisionError:
>>>     print("No se puede dividir entre 0")
Error en la operación
```

Recuerda entonces:

- El orden de las excepciones importa.
- No pongas excepciones más generales antes que otras más concretas ya que esto hará que la última sea inalcanzable e inútil.

Por otra parte, si se desean manejar dos o más excepciones de la misma manera, se puede utilizando el bloque try-except del siguiente modo:

```
>>> try:
>>>     x = int(input("Escriba un número"))
>>>     y = 200 / x
>>> except (ZeroDivisionError, ValueError):
>>>     print("Ha ocurrido un error")
```

Hay dos palabras reservadas más relacionadas con el bloque try-except que son **else** y **finally**. La palabra reservada else se debe utilizar después de las instrucciones except y sirve para definir un código que se va a ejecutar sólo si no ha ocurrido ningún error. Veamos un ejemplo:

```
>>> try:
>>>     x = int(input("Escriba un número"))
>>>     y = 200 / x
>>> except ZeroDivisionError:
>>>     print("No se puede dividir entre 0")
>>> except ValueError:
>>>     print("Debes introducir un número entero")
>>> except:
>>>     print("Alguna otra cosa ha salido mal")
>>> else:
>>>     print("El valor de y es: ", y)
```

En este caso si el usuario introdujese el valor "20", se mostrará por pantalla "El valor de y es 10.0".

La otra palabra reservada es finally que se colocará al final de todas las expresiones except y de else si también se utiliza. Esta palabra reservada sirve definir un código que se va a ejecutar tanto si ha ocurrido un error (se capture o no) como si no ha ocurrido ningún error.

```
>>> try:
>>>     x = int(input("Escriba un número"))
>>>     y = 200 / x
>>> except ZeroDivisionError:
>>>     print("No se puede dividir entre 0")
>>> except ValueError:
>>>     print("Debes introducir un número entero")
>>> except:
>>>     print("Alguna otra cosa ha salido mal")
>>> else:
>>>     print("El valor de y es: ", y)
>>> finally:
>>>     print("FINAL")
```

En este caso si el usuario introdujese el valor "20", se mostrará por pantalla "El valor de y es 10.0" y "FINAL" y si el usuario introdujese el valor "0", se mostrará "No se puede dividir entre 0" y "FINAL".

3.3. Excepciones: Lanzar una excepción

Ya hemos visto cómo capturar una excepción, pero ¿sería posible lanzar de forma consciente una excepción? Lo es. Python para ello proporciona la instrucción **raise** que genera la excepción especificada. Su sintaxis es:

raise excepción

Donde excepción puede ser una nueva excepción creada, por ejemplo, si queremos lanzar una excepción en nuestro código:

```
>>> try:
>>>     raise ZeroDivisionError
>>> except ZeroDivisionError:
>>>     print("No se puede dividir entre 0")
No se puede dividir entre 0

```

O puede ser ejecutada dentro de la cláusula except para relanzar la excepción, en ese caso se puede utilizar también raise sin nada más:

```
>>> try:
>>>     y = 200 / 0
>>> except ZeroDivisionError:
>>>     print("No se puede dividir entre 0")
>>>     raise
No se puede dividir entre 0
Traceback
ZeroDivisionError: división by zero

```

En este último ejemplo se muestra como recoger el objeto de excepción con as ex y cómo con el raise lo volveríamos a lanzar.

```
>>> try:
>>>     y = 200 / 0
>>> except ZeroDivisionError as ex:
>>>     print("No se puede dividir entre 0")
>>>     raise ex
No se puede dividir entre 0
Traceback
ZeroDivisionError: división by zero

```

4. Generadores y Cierres

Una vez visto el tema de excepciones, vamos a ver otro tema que también es importante conocer de cara a la certificación y que son los generadores y cierres.

4.1. Generadores

Comencemos con los generadores. ¿Qué es un **generador**? Un generador de Python es un **fragmento de código especializado capaz de producir una serie de valores y controlar el proceso de iteración**. Por ello a menudo a los generadores también se les denomina iteradores.

Parece un concepto complicado, pero, de hecho, en muchas lecciones anteriores hemos estado trabajando con generadores sin saberlo. Este es el caso de la función range. La función range() es un generador e iterador.

Y, ¿qué es un **iterador**? Un iterador es un objeto que cumple el protocolo iterador. El protocolo iterador es una forma en que un objeto debe comportarse para ajustarse a las reglas impuestas por las sentencias *for* e *in*. Para cumplir este protocolo, un iterador debe proporcionar dos métodos:

- __iter__() -> Este método debe devolver el objeto en sí y se invoca una sola vez.
- __next__() → Este método debe devolver el siguiente valor de la serie, es decir, cada vez que se llama debe ir devolviendo el siguiente valor de la serie y cuando ya no haya más valores deberá lanzar una excepción de tipo StopIteration. Este método es el que va llamando la sentencia for-in.

Vamos a ver cómo crear y utilizar un objeto iterador con un ejemplo.

```
1.      >>> class MisPotencias:
2.      >>>     def __init__(self, base, maxima_potencia):
3.      >>>         self.base = base
4.      >>>         self.maxima_potencia = maxima_potencia
5.      >>>         self.__i = -1
6.      >>>     def __iter__(self):
7.      >>>         print("__iter__")
8.      >>>         return self
9.      >>>     def __next__(self):
10.     >>>         print("__next__")
11.     >>>         self.__i += 1
12.     >>>         if self.__i > self.maxima_potencia:
13.     >>             print("Lanzo StopIteration")
14.     >>>             raise StopIteration
15.     >>>         return self.base ** self.__i
16.     >>>
17.     >>> for i in MisPotencias(2,5):
18.     >>>     print(i)
19.     __iter__
20.     __next__
21.     1
22.     __next__
23.     2
24.     __next__
25.     4
26.     __next__
27.     8
28.     __next__
29.     16
30.     __next__
31.     32
32.     __next__
33.     Lanzo StopIteration
34.
```

En el ejemplo se ha creado una clase iteradora llamada MisPotencias que genera una serie de valores que corresponden a las potencias desde 0 hasta maxima_potencia utilizando como base y maxima_potencia las indicadas en el constructor. Vamos a ver el código en más detalle:

- En las líneas 2-5, se declara el constructor que recibe la base y la maxima_potencia que se van a utilizar y que inicializa la variable __i que será el exponente que se irá utilizando para general la serie.
- En las líneas 6-8, se declara el método __iter__ que devuelve el propio objeto iterador.

- En las líneas 9-14, se declara el método __next__ que irá generando la serie de potencias. En este método se comprueba si el exponente, __i, ha llegado al máximo que el usuario había indicado y en ese caso lanza una excepción de tipo StopIteration. Si no ha llegado al máximo devuelve el valor correspondiente de la potencia.
- En las líneas 16-17, se muestra cómo se crea y se llama al objeto iterador.
- En las siguientes líneas se muestra la salida por pantalla. En esta salida por pantalla se observa que lo primero que se ha ejecutado en el for es la función __iter__ y que esta función sólo se ha ejecutado una vez. Posteriormente, se ve que se ha ido llamando sucesivamente a la función __next__ que ha ido devolviendo las sucesivas potencias desde 0 hasta 5 que es el valor que se había indicado y la siguiente vez que se llama cómo lanza la excepción StopIteration para terminar con la iteración.

Como se puede observar, la implementación del protocolo iterador es sencilla de entender y de utilizar. Además del protocolo iterador, Python ofrece otra manera más efectiva y elegante de escribir iteradores. Esta herramienta es la palabra reservada **yield**. Esta palabra reservada es similar a la palabra reservada return, pero con la diferencia de que si se usa yield en una función convierte a la función en un generador. Veámoslo con un ejemplo:

```
1.      >>> def mis_potencias(base, maxima_potencia):
2.      >>>     for i in range(0, maxima_potencia + 1):
3.      >>>         print("Dentro")
4.      >>>         yield base ** i
5.      >>>
6.      >>> for i in mis_potencias(2,5):
7.      >>>    print(i)
8.      Dentro
9.      1
10.     Dentro
11.     2
12.     Dentro
13.     4
14.     Dentro
15.     8
16.     Dentro
17.     16
18.     Dentro
19.     32
20.
```

En este ejemplo se ha creado un generador que hace lo mismo que el generador del ejemplo anterior, pero como podemos ver el número de líneas necesarias para escribirlo es bastante inferior. Veamos entonces el comportamiento de yield:

- Al utilizar yield en una función en vez de return, la función se convierte en un generador y devuelve los elementos bajo demanda.
- Al llamar a la función lo que devuelve es el objeto iterador, no la serie que esperamos del generador.
- Cada vez que devuelve un valor no pierde el estado de la función.
- La función ya no podrá invocarse explícitamente ya que no es una función sino un objeto generador.

4.2. Cierres

Para terminar con esta sección, vamos a ver qué son los **cierres**. Cierres es una **técnica que permite almacenar valores a pesar de que el contexto en el que se crearon ya no existe**. Este concepto suena y es un poco complicado de entender. Pero veámoslo mejor con un ejemplo:

```
1.      >>> def mi_funcion(a):
2.      >>>     loc = 2 * a
3.      >>>
4.      >>> var = 5
5.      >>> mi_funcion(var)
6.      >>> print(var)
7.      >>> print(loc)
8.
9.      5
10.     Traceback
11.     NameError: name 'loc' is not defined
12.
```

Como se puede observar en el ejemplo, una vez llamada a la función mi_funcion() ya no podremos acceder al valor de loc ya que loc no es accesible fuera de la función. Para que se pudiese conservar el valor, sería necesario convertir la función en un cierre. Veamos cómo se haría esto:

```
1.      >>> def mi_funcion(a):
2.      >>>     loc = 2 * a
3.      >>>     def interior():
4.      >>>         return loc
5.      >>>     return interior
6.      >>>
7.      >>> var = 5
8.      >>> fun = mi_funcion(var)
9.      >>> print(var)
10.     >>> print(fun())
11.
12.     5
13.     10
14.
```

Vamos línea por línea estudiando este código:

- Líneas 1-5: Se ha definido la función:
  - Línea 2: Se realiza la operación deseada.
  - Línea 3-4: Se define una función dentro de mi_funcion() que me devuelve el valor que quiero poder acceder posteriormente.
  - Línea 5: No se devuelve el valor, sino la función. Es importante ver que no se están utilizando los paréntesis al devolver la función ya que si se utilizasen se devolvería el valor no el objeto función.
- Línea 8: Se llama a la función y lo que se almacena en fun será el objeto función no el valor.
- Línea 10: Se llama al objeto función, esta vez utilizando los paréntesis para poder acceder al valor almacenado por esta función.
- Indicar que la función interior podría tener parámetros y en ese caso cuando se llamase a fun() habría que introducir el mismo número de parámetros que tuviese definidos la función interior.

5. FUNCIONES LAMBDA

Para seguir con la lección, vamos a ver qué es una función **lambda**. Una función lambda es una **función sin nombre**, es decir, es una función anónima. Este tipo de funciones se declara de la siguiente manera:

lambda parámetros: expresión

Veamos un ejemplo:

```
1.      >>> fun = lambda x,y: x * y
2.      >>> print(fun(2,3))
3.      6
4.
```

En este ejemplo podemos ver cómo se declara una función lambda que recibe dos parámetros y devuelve el resultado de su multiplicación. Al declararla, la asignamos a la variable fun y posteriormente en la siguiente línea al ejecutar fun(parámetros) la llamamos.

Las funciones lambda también se pueden declarar sin argumentos, pero siempre tienen que devolver un valor:

```
1.      >>> fun = lambda: 2
2.      >>> print(fun())
3.      2
4.
```

El concepto de función lambda es sencillo de entender y hay que conocerlo ya que este tipo de funciones pueden aparecer en el examen de certificación, especialmente como parte de otras preguntas.

Un uso muy común de este tipo de funciones es con la función **map**(). La función map() tomas dos argumentos: una función y una lista y aplica la función pasada a todos los elementos de la lista pasada devolviendo un iterador con los resultados. Veamos un ejemplo:

```
>>> lista1 = [1, 2, 3, 4, 5]
>>> lista2 = list(map(lambda x: 2 * x, lista1))
>>> print(lista2)
[2, 4, 6, 8, 10]
```

Otra función también muy utilizada con la función lambda es la función **filter**(). La función filter espera los mismos argumentos que la función map, pero hace algo diferente. Filter su segundo argumento y devuelve un iterador con los elementos que cumplen la expresión definida en la función lambda. Veamos un ejemplo:

```
>>> lista1 = [1, 2, 3, 4, 5]
>>> lista2 = list(filter(lambda x: x % 2 == 0,
                         lista1))
>>> print(lista2)
[2, 4]
```

Como vemos el iterador que ha devuelto sólo contiene los datos de lista1 que son pares ya que es la expresión indicada en la función lambda.

6. PROCESANDO ARCHIVOS

Para terminar la lección y el contenido de la asignatura, vamos a ver brevemente cómo trabajar con ficheros en Python.

Cualquier programa en Python no se comunica con los archivos directamente, sino a través de algunas entidades abstractas que se denominan **streams** (una especie de canal).

Python proporciona un conjunto de funciones y métodos para realizar ciertas operaciones con el stream que afectan los archivos reales. Es decir, las operaciones realizadas con el stream abstracto reflejan las actividades relacionadas con el archivo físico.

6.1. Abriendo el archivo

La primera operación necesaria para trabajar con el stream es conectar o vincular el stream con el archivo físico. Esta operación se denomina **abrir el archivo**.

Al abrir el archivo se debe declarar la manera o modo en que se va a procesar, existiendo tres modos diferentes:

- **Modo lectura**: Un stream abierto en este modo permite sólo operaciones de lectura, por lo que si se intenta escribir sobre un stream abierto en este modo provocará una excepción.
- **Modo escritura**: Un stream abierto en este modo permite sólo operaciones de escritura, por lo que si se intenta escribir sobre un stream abierto en este modo provocará una excepción.
- **Modo actualizar**: Un stream abierto en este modo permite tanto lectura como escritura.

Abrir el stream se realiza mediante la función **open**:

```
stream = open(fichero, mode='r', encoding=None)
```

Donde el parámetro fichero especifica el nombre del archivo que se asociará al stream, si no puede abrir el stream la función lanzará una excepción FileNotFoundError. El segundo parámetro mode especifica el modo de apertura utilizado para el stream. Y el tercer parámetro especifica el tipo de codificación. Veamos algunos modos de apertura:

- Modo de apertura "**r**" - Lectura: Esto indica que el stream será abierto en modo lectura, por lo que el archivo asociado debe existir y tiene que ser legible, si no se lanzará una excepción.
- Modo de apertura "**w**" – Escritura: Esto indica que el stream será abierto en modo escritura por lo que el archivo asociado puede existir o no. Si no se pudiese escribir se lanzará una excepción. Si existe se borrará todo su contenido previo.
- Modo de apertura "**a**" – Adjuntar: Esto indica que el stream será abierto en modo adjuntar. El archivo asociado no necesita existir, si existe se empezará escribiendo al final del archivo.
- Modo de apertura "**r+**" - leer y adjuntar: Esto significa que el stream será abierto en modo leer y actualizar. Por lo que debe existir y tiene que ser escribible.
- Modo de apertura "**w+**" - Escribir y actualizar: Esto significa que el stream será abierto en modo escribir y actualizar. El fichero asociado no necesita existir, si existe se empezará escribiendo al inicio del fichero.

Además, el fichero se puede abrir en modo texto (por defecto) o en modo binario. Para indicar que se quiere abrir en modo binario se deberá añadir una "**b**" al final del modo de apertura elegido, si por el contrario se quiere abrir en modo texto, se puede añadir una "**t**" al final del modo de apertura elegido o no poner nada. Por último, también se puede abrir un archivo para su creación exclusiva, esto significa que si el archivo ya existe se lanzará una excepción. Para abrirlo en este modo se usará el modo de apertura "**x**".

6.2. Cerrando el archivo

Como se ha indicado la primera operación que debe hacerse con un stream es abrirlo. Mientras que la última operación que deberá hacerse será cerrarlo. Para ello se utiliza la función **close**:

stream.close()

```
>>> try:
>>>     stream = open("/home/usuario/mi_fichero.txt",
                      "r")
>>>     # Se procesa el fichero
>>>     stream.close()
>>> except Exception:
>>>     print("No se puede abrir el fichero")
```

En el ejemplo se muestra cómo se abriría un stream asociado al fichero mi_fichero.txt en modo lectura y en modo texto y cómo posteriormente se cerraría. Si el fichero no existiese se mostraría el mensaje "No se puede abrir el fichero".

Una vez que hemos visto cómo se abre y se cierra un stream, nos faltaría ver cómo procesar el fichero, es decir como leerlo o escribirlo. Este procesamiento dependerá de si se ha abierto en modo texto o binario ya que dependiendo el modo el stream será de un tipo de objeto u otro y tendrá unos métodos u otros.

6.3. Procesando ficheros de texto

Comencemos viendo resumidamente cómo se procesa un fichero de texto y para ello vamos a comenzar viendo las principales funciones de lectura:

La función **read**() lee todo el contenido del fichero y lo devuelve en una cadena:

```
>>> try:
>>>     stream = open("/home/usuario/mi_fichero.txt",
                      "r")
>>>     print(stream.read())
>>>     stream.close()
>>> except Exception:
>>>     print("No se puede abrir el fichero")
Contenido
es este.
```

La función **read**() también permite un parámetro para indicar el número de caracteres a leer cada vez que se llama. Cuando ya no hay más caracteres qué leer devuelve una cadena vacía.

```
>>> try:
>>>     stream = open("/home/usuario/mi_fichero.txt",
                      "r")
>>>     ch = stream.read(1)
>>>     while ch ¡= '':
>>>         print(ch, end='')
>>>         ch = stream.read(1)
>>>     stream.close()
>>> except Exception:
>>>     print("No se puede abrir el fichero")
Contenido
es este.
```

La función **readline**() permite leer el fichero de línea en línea y cuando ya no hay más líneas que leer devuelve una cadena vacía:

```
>>> try:
>>>     stream = open("/home/usuario/mi_fichero.txt",
                      "r")
>>>     lin = stream.readline()
>>>     while lin ¡= '':
>>>         print(lin)
>>>         lin = stream.readline()
>>>     stream.close()
>>> except Exception:
>>>     print("No se puede abrir el fichero")
Contenido
es este.
```

La función **readlines**() permite leer el fichero y devuelve una lista con cada una de las líneas de texto:

```
>>> try:
>>>     stream = open("/home/usuario/mi_fichero.txt",
                      "r")
>>>     lines = stream.readlines()
>>>     print(lines)
>>>     stream.close()
>>> except Exception:
>>>     print("No se puede abrir el fichero")
["Contenido", "es este."]
```

El objeto stream es además un objeto iterable, por lo que se puede iterar para leer su contenido:

```
>>> try:
>>>     stream = open("/home/usuario/mi_fichero.txt",
                      "r")
>>>     for lin in stream:
>>>         print(lin)
>>>     stream.close()
>>> except Exception:
>>>     print("No se puede abrir el fichero")
Contenido
 es este.
```

Por otra parte, para escribir un fichero de texto tenemos las siguientes funciones:

La función **write**() permite escribir una cadena de texto:

```
>>> try:
>>>     stream = open("/home/usuario/mi_fichero.txt",
                      "w")
>>>     stream.write("Mi cadena")
>>>     stream.close()
>>> except Exception:
>>>     print("No se puede escribir el fichero")
```

La función **writelines**() permite escribir una lista de cadenas de texto:

```
>>> try:
>>>     stream = open("/home/usuario/mi_fichero.txt",
                      "w")
>>>     stream.writelines(["Mi cadena", "Mi otra",
                          "cadena"])
>>>     stream.close()
>>> except Exception:
>>>     print("No se puede escribir el fichero")
```

6.4. Procesando ficheros binarios

Para terminar este capítulo y por lo tanto la lección, vamos a ver resumidamente cómo se procesa un fichero binario. Para ello, comencemos definiendo qué es un **bytearray**. Un bytearray es una clase especializada de Python para almacenar datos amorfos, es decir, datos que no tienen una forma específica, una serie de bytes. Por lo que un **bytearray es un contenedor que contiene bytes**. Para crear un bytearray lo haremos con la siguiente instrucción:

data = bytearray(num)

Donde num será el número de bytes que va a contener. Con esta instrucción, el contenedor se rellenará con ceros. El contenedor es mutable y cualquiera de sus elementos se puede tratar como un valor entero.

Esta clase es muy importante para poder trabajar con ficheros binario. Una vez vista, vamos a comenzar viendo cómo se lee este tipo de ficheros:

La función **readinto**() lee el contenido del fichero binario y lo guarda en la estructura bytearray dada. Esta función devuelve el número de bytes leídos correctamente, además sólo lee hasta que el bytearray se llena, una vez se llena detiene la lectura.

```
>>> try:
>>>     data = bytearray(5)
>>>     stream = open("/home/usuario/mi_fichero.bin",
                      "rb")
>>>     num = stream.readinto(data)
>>>     stream.close()
>>>     for bin in data:
>>>         print(hex(bin), end=' ')
>>> except Exception:
>>>     print("No se puede abrir el fichero")
0xa 0xb 0x3 0x10 0x1
```

Como podemos observar con estas instrucciones se habrás leído 5 bytes del fichero, rellenando la estructura bytearray. Si el fichero tiene menos de 5 bytes, se habrán leído sólo los bytes que tiene el fichero y el resto de posiciones de bytearray estará a 0. Además, en ese caso num será menor a 5 ya que indicará el número de bytes leído.

La función **read**() lee todo el contenido del fichero. También se puede utilizar con un argumento para indicar el número máximo de bytes a leer.

```
>>> try:
>>>     stream = open("/home/usuario/mi_fichero.bin",
                      "rb")
>>>     data = bytearray(stream.read())
>>>     stream.close()
>>>     for bin in data:
>>>         print(hex(bin), end=' ')
>>> except Exception:
>>>     print("No se puede abrir el fichero")
0xa 0x1 0x4 0x10 0x5
```

Como se puede observar la sintaxis de esta función es bastante diferente a la anterior ya que no se le pasa el bytearray a rellenar, sino que su salida se convierte a bytearray.

Por último, para escribir un fichero binario tenemos la siguiente función:

- La función **write**() donde se le pasa el bytearray a escribir. Esta función devuelve el número de bytes correctamente escrito.

```
>>> try:
>>>     stream = open("/home/usuario/mi_fichero.bin",
                      "wb")
>>>     data = bytearray(10)
>>>     for i in range(len(data)):
>>>         data[i] = 10 + 1
>>>     num = stream.write(data)
>>>     print(num)
>>>     stream.close()
>>> except Exception:
>>>     print("No se puede abrir el fichero")
10
```

7. Puntos Clave

Como en cada lección, en este apartado resumimos las conclusiones principales obtenidas del estudio de la misma:

- Esta lección ha comenzado estudiando qué es un módulo y las tres formas diferentes de importarlo, explicando las implicaciones que tiene cada una de ellas. Además, se ha visto cómo es posible agrupar estos módulos en paquetes.

- Se ha estudiado cómo controlar los posibles errores, viendo las diferentes formas de capturar las excepciones y sus particularidades.

- Se ha estudiado la jerarquía que siguen las excepciones y sus implicaciones.

- Se ha estudiado cómo lanzar una excepción de diferentes formas y en diferentes contextos.

- Se han estudiado los conceptos de generado e iterador y se ha visto cómo crearlos usando el protocolo de iteración o la sentencia yield.

- Se ha estudiado el concepto de cierre y se ha visto cómo se crean y usan.

- Se ha visto cómo definir una función lambda y su utilidad dentro de las funciones map y filter.

- Se ha visto cómo se trabaja en Python con ficheros de texto y binarios, viendo los diferentes modos de apertura y las principales funciones para su manipulación.